10/015,131

WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Friday, March 26, 2004

Hide?	Set Name	Query	Hit Count
		DB=USPT; PLUR=NO; OP=ADJ	
□	L29	L28 and (l10 or l7 or l12 or l3 or l4 or l5)	11
□	L28	L27 and l16	12
□	L27	(L23 or l22) and generation$1	503
□	L26	L23 and l22 and generation$1	0
□	L25	L23 and l22 and generation$1 and l11	0
□	L24	L23 and l22 and generation$1 and l16	0
□	L23	remembered adj1 set	50
□	L22	previous adj1 version$1	1354
□	L21	L20 and l1 and (l8 or l9)	1
□	L20	page$1 near5 generation	1192
□	L19	(4797810)![pn]	1
□	L18	L17 and (l2 or l3 or l6 or l7 or l8 or l9 or l10 or l11 or l12)	2
□	L17	L16 and l5	2
□	L16	707/202.ccls.	642
□	L15	l5 and (l1 or l2 or l6 or l11)	7
□	L14	l4 and l5	0
□	L13	l1 and l4 and l5	0
□	L12	disk	248383
□	L11	generation-specific or (generation adj1 specific)	672
□	L10	memory near5 (central or cells)	62004
□	L9	reconstruct$4 near5 set	2271
□	L8	reconstruct$4	49526
□	L7	garbage	9528
□	L6	garbage adj1 collection	1305
□	L5	mature adj1 generation	12
□	L4	first adj1 generation	5449
□	L3	back-up	24283
□	L2	recover$4 near5 database	922
□	L1	(707/202 or 707/206).ccls.	978

END OF SEARCH HISTORY

IEEE Xplore®
RELEASE 1.6

Welcome
United States Patent and Trademark Office

Your search matched **0** of **1015452** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevanc Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

first and mature and generation and database and reco [Search]

□ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

Results:
No documents matched your query.

IEEE HOME | SEARCH IEEE | SHOP | WEB ACCOUNT | CONTACT IEEE

IEEE

Membership Publications/Services Standards Conferences Careers/Jobs

IEEE Xplore®
RELEASE 1.6

Welcome
United States Patent and Trademark Office

Help FAQ Terms IEEE Peer Review

Quick Links

» Se

Welcome to IEEE Xplore®

- ○- Home
- ○- What Can I Access?
- ○- Log-out

Tables of Contents

- ○- Journals & Magazines
- ○- Conference Proceedings
- ○- Standards

Search

- ○- By Author
- ○- Basic
- ○- Advanced

Member Services

- ○- Join IEEE
- ○- Establish IEEE Web Account
- ○- Access the IEEE Member Digital Library

Your search matched **1** of **1015452** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevanc Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

mature and generation and database and (recovery or [Search]

□ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

1 **Program plan recognition for Year 2000 tools**
van Deursen, A.; Woods, S.; Quilici, A.;
Reverse Engineering, 1997. Proceedings of the Fourth Working Conference or
Oct. 1997
Pages:124 - 133

[Abstract] [PDF Full-Text (772 KB)] **IEEE CNF**

Home | Log-out | Journals | Conference Proceedings | Standards | Search by Author | Basic Search | Advanced Search | Join IEEE | Web Account | New this week | OPAC Linking Information | Your Feedback | Technical Support | Email Alerting | No Robots Please | Release Notes | IEEE Online Publications | Help | FAQ| Terms | Back to Top

IEEE HOME | SEARCH IEEE | SHOP | WEB ACCOUNT | CONTACT IEEE

IEEE

Membership Publications/Services Standards Conferences Careers/Jobs

IEEE Xplore®
RELEASE 1.6

Welcome
United States Patent and Trademark Office

Help FAQ Terms IEEE Peer Review

Quick Links

» Se

Welcome to IEEE Xplore®

- ○- Home
- ○- What Can I Access?
- ○- Log-out

Tables of Contents

- ○- Journals & Magazines
- ○- Conference Proceedings
- ○- Standards

Search

- ○- By Author
- ○- Basic
- ○- Advanced

Member Services

- ○- Join IEEE
- ○- Establish IEEE Web Account
- ○- Access the IEEE Member Digital Library

Your search matched **0** of **1015452** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevance Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

'mature generation' and garbage Search

□ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

Results:
No documents matched your query.

Home | Log-out | Journals | Conference Proceedings | Standards | Search by Author | Basic Search | Advanced Search | Join IEEE | Web Account | New this week | OPAC Linking Information | Your Feedback | Technical Support | Email Alerting | No Robots Please | Release Notes | IEEE Online Publications | Help | FAQ| Terms | Back to Top

IEEE HOME | SEARCH IEEE | SHOP | WEB ACCOUNT | CONTACT IEEE

Membership Publications/Services Standards Conferences Careers/Jobs

IEEE Xplore® RELEASE 1.6

Welcome
United States Patent and Trademark Office

Help FAQ Terms IEEE Peer Review

Quick Links

» Se

Welcome to IEEE Xplore®

- ○- Home
- ○- What Can I Access?
- ○- Log-out

Tables of Contents

- ○- Journals & Magazines
- ○- Conference Proceedings
- ○- Standards

Search

- ○- By Author
- ○- Basic
- ○- Advanced

Member Services

- ○- Join IEEE
- ○- Establish IEEE Web Account
- ○- Access the IEEE Member Digital Library

Your search matched **1** of **1015452** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevanc** **Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

mature and generation and garbage and collection [Search]

□ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

1 **Starting with termination: a methodology for building distributed garbage collection algorithms**
Blackburn, S.M.; Hudson, R.L.; Morrison, R.; Moss, J.E.B.; Munro, D.S.; Zigm J.;
Computer Science Conference, 2001. ACSC 2001. Proceedings. 24th Australasian , 29 Jan-4 Feb 2001
Pages:20 - 28

[Abstract] [PDF Full-Text (788 KB)] **IEEE CNF**

Home | Log-out | Journals | Conference Proceedings | Standards | Search by Author | Basic Search | Advanced Search | Join IEEE | Web Account | New this week | OPAC Linking Information | Your Feedback | Technical Support | Email Alerting | No Robots Please | Release Notes | IEEE Online Publications | Help | FAQ| Terms | Back to Top

First Hit Fwd Refs

□ Generate Collection Print

L15: Entry 1 of 7 File: USPT Mar 4, 2003

DOCUMENT-IDENTIFIER: US 6529919 B1
TITLE: Incremental class unloading in a train-algorithm-based garbage collector

Parent Case Text (2):
This application is related to U.S. patent application Ser. No. 09/377,349 of Alexander T. Garthwaite for Popular-Object Handling in a Train-Algorithm-Based Garbage Collector, U.S. patent application Ser. No. 09/377,473 of Alexander T. Garthwaite for Scalable-Remembered-Set Garbage Collection, U.S. patent application Ser. No. 09/377,137 of Garthwaite et al. for Reduced-Cost Remembered-Set Processing in a Train-Algorithm-Based Garbage Collector, U.S. patent application Ser. No. 09/377,555 of Alexander T. Garthwaite for Train-Algorithm-Based Garbage Collector Employing Fixed-Size Remembered Sets, U.S. patent application Ser. No. 09/377,289 of Alexander T. Garthwaite for Train-Algorithm-Based Garbage Collector Employing Reduced Oversize-Object Threshold, U.S. patent application Ser. No. 09/377,654 of Garthwaite et al. for a Train-Algorithm-Based Garbage Collector Employing Farthest-Forward-Car Indicator, all of which were filed on Aug. 19, 1999, are assigned to the instant application's assignee, and are hereby incorporated in their entirety by reference.

Brief Summary Text (3):
The present invention is directed to memory management. It particularly concerns what has come to be known as "garbage collection."

Brief Summary Text (16):
A way of reducing the likelihood of such leaks and related errors is to provide memory-space reclamation in a more-automatic manner. Techniques used by systems that reclaim memory space automatically are commonly referred to as "garbage collection." Garbage collectors operate by reclaiming space that they no longer consider "reachable." Statically allocated objects represented by a program's global variables are normally considered reachable throughout a program's life. Such objects are not ordinarily stored in the garbage collector's managed memory space, but they may contain references to dynamically allocated objects that are, and such objects are considered reachable. Clearly, an object referred to in the processor's call stack is reachable, as is an object referred to by register contents. And an object referred to by any reachable object is also reachable.

Brief Summary Text (18):
Garbage-collection mechanisms can be implemented by various parts and levels of a computing system. One approach is simply to provide them as part of a batch compiler's output. Consider FIG. 2's simple batch-compiler operation, for example. A computer system executes in accordance with compiler object code and therefore acts as a compiler 20. The compiler object code is typically stored on a medium such as FIG. 1's system disk 17 or some other machine-readable medium, and it is loaded into RAM 14 to configure the computer system to act as a compiler. In some cases, though, the compiler object code's persistent storage may instead be provided in a server system remote from the machine that performs the compiling. The electrical signals that carry the digital data by which the computer systems exchange that code are exemplary forms of carrier waves transporting the information.

Brief Summary Text (19):
The input to the compiler is the application source code, and the end product of the compiler process is application object code. This object code defines an application 21, which typically operates on input such as mouse clicks, etc., to generate a display or some other type of output. This object code implements the relationship that the programmer intends to specify by his application source code. In one approach to garbage collection, the compiler 20, without the programmer's explicit direction, additionally generates code that automatically reclaims unreachable memory space.

Brief Summary Text (20):
Even in this simple case, though, there is a sense in which the application does not itself provide the entire garbage collector. Specifically, the application will typically call upon the underlying operating system's memory-allocation functions. And the operating system may in turn take advantage of various hardware that lends itself particularly to use in garbage collection. So even a very simple system may disperse the garbage-collection mechanism over a number of computer-system layers.

Brief Summary Text (21):
To get some sense of the variety of system components that can be used to implement garbage collection, consider FIG. 3's example of a more complex way in which various levels of source code can result in the machine instructions that a processor executes. In the FIG. 3 arrangement, the human applications programmer produces source code 22 written in a high-level language. A compiler 23 typically converts that code into "class files." These files include routines written in instructions, called "byte codes" 24, for a "virtual machine" that various processors can be configured to emulate. This conversion into byte codes is almost always separated in time from those codes' execution, so FIG. 3 divides the sequence into a "compile-time environment" 25 separate from a "run-time environment" 26, in which execution occurs. One example of a high-level-language environment for which compilers are available to produce such virtual-machine instructions is the Java.TM. platform. (Java is a trademark or registered trademark of Sun Microsystems, Inc., in the Unites States and other countries.)

Brief Summary Text (23):
In most implementations, much of the virtual machine's action in executing these byte codes is most like what those skilled in the art refer to as "interpreting," and FIG. 3 shows that the virtual machine includes an "interpreter" 28 for that purpose. The resultant instructions typically invoke calls to a run-time system 29, which handles matters such as loading new class files as they are needed and, of particular interest in the present connection, performing garbage collection.

Brief Summary Text (25):
The arrangement of FIG. 3 differs from FIG. 2 in that the compiler 23 for converting the human programmer's code does not contribute to providing the garbage-collection function; that results largely from the virtual machine 27's operation. Although the FIG. 3 arrangement is a popular one, it is by no means universal, and many further implementation types can be expected. Proposals have even been made to implement the virtual machine 27's behavior in a hardware processor, in which case the hardware itself would provide some or all of the garbage-collection function.

Brief Summary Text (26):
In short, garbage collectors can be implemented in a wide range of combinations of hardware and/or software. As is true of most of the garbage-collection techniques described in the literature, the invention to be described below is applicable to most such systems.

Brief Summary Text (27):
By implementing garbage collection, a computer system can greatly reduce the

occurrence of memory leaks and other software deficiencies in which human programming frequently results. But it can also have significant adverse performance effects if it is not implemented carefully. To distinguish the part of the program that does "useful" work from that which does the garbage collection, the term mutator is sometimes used in discussions of these effects; from the collector's point of view, what the mutator does is mutate active data structures' connectivity.

Brief Summary Text (28):
Some garbage-collection approaches rely heavily on interleaving garbage-collection steps among mutator steps. In one type of garbage-collection approach, for instance, the mutator operation of writing a reference is followed immediately by garbage-collector steps used to maintain a reference count in that object's header, and code for subsequent new-object storage includes steps for finding space occupied by objects whose reference count has fallen to zero. Obviously, such an approach can slow mutator operation significantly.

Brief Summary Text (29):
Other approaches therefore interleave very few garbage-collector-related instructions into the main mutator process but instead interrupt it from time to time to perform garbage-collection cycles, in which the garbage collector finds unreachable objects and reclaims their memory space for reuse. Such an approach will be assumed in discussing FIG. 4's depiction of a simple garbage-collection operation. Within the memory space allocated to a given application is a part 40 managed by automatic garbage collection. In the following discussion, this will be referred to as the "heap," although in other contexts that term refers to all dynamically allocated memory. During the course of the application's execution, space is allocated for various objects 42, 44, 46, 48, and 50. Typically, the mutator allocates space within the heap by invoking the garbage collector, which at some level manages access to the heap. Basically, the mutator asks the garbage collector for a pointer to a heap region where it can safely place the object's data. The garbage collector keeps track of the fact that the thus-allocated region is occupied. It will refrain from allocating that region in response to any other request until it determines that the mutator no longer needs the region allocated to that object.

Brief Summary Text (31):
A typical approach to garbage collection is therefore to identify all reachable objects and reclaim any previously allocated memory that the reachable objects do not occupy. A typical garbage collector may identify reachable objects by tracing references from the root set 52. For the sake of simplicity, FIG. 4 depicts only one reference from the root set 52 into the heap 40. (Those skilled in the art will recognize that there are many ways to identify references, or at least data contents that may be references.) The collector notes that the root set points to object 42, which is therefore reachable, and that reachable object 42 points to object 46, which therefore is also reachable. But those reachable objects point to no other objects, so objects 44, 48, and 50 are all unreachable, and their memory space may be reclaimed.

Brief Summary Text (32):
To avoid excessive heap fragmentation, some garbage collectors additionally relocate reachable objects. FIG. 5 shows a typical approach. The heap is partitioned into two halves, hereafter called "semi-spaces." For one garbage-collection cycle, all objects are allocated in one semi-space 54, leaving the other semi-space 56 free. When the garbage-collection cycle occurs, objects identified as reachable are "evacuated" to the other semi-space 56, so all of semi-space 54 is then considered free. Once the garbage-collection cycle has occurred, all new objects are allocated in the lower semi-space 56 until yet another garbage-collection cycle occurs, at which time the reachable objects are evacuated back to the upper semi-space 54.

Brief Summary Text (34):
In one sense, the approach of interrupting the mutator occasionally for garbage collection can increase an application's responsiveness, because the main mutator operation ordinarily proceeds relatively unburdened by garbage-collection overhead. In interactive systems, moreover, interruptions for garbage collection can sometimes be scheduled opportunistically so as to reduce the like-lihood that they will result in much overall speed reduction. Garbage-collection an be triggered when the system is waiting for user input, for instance.

Brief Summary Text (35):
So it may often be true that the garbage-collection operation's effect on performance can depend less on the total collection time than on when collections actually occur. But another factor that often is even more determinative is the duration of any single collection cycle, i.e., how long the mutator must remain quiescent at any one time. In an interactive system, for instance, a user may never notice hundred-millisecond interruptions for garbage collection, whereas most users would find interruptions lasting for two seconds to be annoying. Many garbage collectors therefore operate incrementally. That is, they perform less than a complete collection in any single interruption of the main application.

Brief Summary Text (40):
Of course, the card-table approach is only one of many that can be employed to detect inter-generational pointers. Indeed, it is typical for an individual garbage collector to use more than one approach. Although there is no reason in principle to favor any particular number of generations, and although FIG. 6 shows three, most generational garbage collectors have only two generations, of which one is the young generation and the other is the mature generation. Moreover, although FIG. 6 shows the generations as being of the same size, a more-typical configuration is for the young generation to be considerably smaller. Finally, although we assumed for the sake of simplicity that collection during a given cycle was limited to only one generation, a more-typical approach is actually to collect the whole young generation at every cycle but to collect the mature one less frequently.

Brief Summary Text (41):
To collect the young generation, it is preferable to employ the card table to identify pointers into the young generation; laboriously scanning the entire mature generation would take too long. On the other hand, since the young generation is collected in every cycle and can therefore be collected before mature-generation processing, it takes little time to scan the few remaining, live objects in the young generation for pointers into the mature generation in order to process that generation. For this reason, the card table will typically be so maintained as only to identify the regions occupied by references into younger generations and not into older ones.

Brief Summary Text (42):
Now, although it typically takes very little time to collect the young generation, it may take more time than is acceptable within a single garbage-collection cycle to collect the entire mature generation. So some garbage collectors may collect the mature generation incrementally; that is, they may perform only a part of the mature generation's collection during any particular collection cycle. Incremental collection presents the problem that, since the generation's objects that are outside a collection cycle's collection set are not processed during that cycle, any such objects that are unreachable are not recognized as unreachable, so collection-set objects to which they refer tend not to be, either.

Brief Summary Text (73):
One shortcoming of the conventional approach to train-algorithm-based garbage collection is that certain types of objects will not end up being identified as unreachable unless measures are taken that compromise the train algorithm's

incremental nature. To appreciate this, consider FIGS. 9A and 9B, which illustrate a typical approach to memory allocation for class information.

Drawing Description Text (6):
FIG. 4, discussed above, is a diagram that illustrates a basic garbage-collection mechanism;

Drawing Description Text (7):
FIG. 5, discussed above, is a similar diagram illustrating that garbage-collection approach's relocation operation;

Current US Original Classification (1):
707/206

Other Reference Publication (1):
Bill Venners, Garbage Collection, Inside the Java 2 Virtual Machine, Chapter 9, parts 1-18, www.artima.com., May 23, 2000.

Other Reference Publication (2):
Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", in Yves Bekkers and Jacques Cohen, editors, Proceedings of International Workshop on Memory Management, vol. 637 of Lecture Notes in Computer Science, 1992 Springer-Verlag.

Other Reference Publication (5):
Andrew W. Appel, "Simple Generational Garbage Collection and Fast Allocation", Software Practice and Experience, 19(2): 171-183, 1989.

Other Reference Publication (6):
Richard Hudson and Amer Diwan, "Adaptive Garbage Collection for Modula-3 and Small Talk" in OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, Oct. 1990, edited by Eric Jul and Niels-Christian Juul.

Other Reference Publication (10):
Urs Holzle, "A Fast Write Barrier for Generational Garbage Collectors" in OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object Oriented Systems, Oct. 1993, edited by Moss, Wilson and Zorn.

Other Reference Publication (11):
Antony L. Hosking and Richard L. Hudson, "Remembered Sets can Also Play Cards" in OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems, Oct. 1993, edited by Moss, Wilson and Zorn.

Other Reference Publication (12):
Jacob Seligmann and Steffen Grarup, "Incremental Mature Garbage Collection Using Train Algorithm", in the European Conference on Object-Oriented Programming 1995 Proceedings. Available at http://www.daimi.aau.dk/jacobse/Papers/.

Other Reference Publication (13):
Steffen Grarup and Jacob Seligmann, "Incremental Mature Garbage Collection" M.Sc. Thesis, avaliable at http://www.daimi.aau.dk/jacobse/Papers/. Aug. 1993.

CLAIMS:

1. A method of garbage collection in which a train-algorithm-managed generation of a garbage-collected heap contains objects located in car sections and is collected in accordance with the train algorithm, in collection cycles for which respective collection sets of the car sections are established, by repeatedly establishing new trains, evacuating from the collection set the objects in the collection set that are referred to from outside the collection set, placing objects thus evacuated into trains containing objects that refer to them, and reclaiming the car sections

that remain in the collection set, wherein the method further includes: A) associating proxy objects in the garbage-collected heap with respective external objects outside the garbage-collected heap; and B) evacuating from the collection set those of its objects that are referred to by external objects associated with proxy objects not in the collection set, the trains into which the objects thus evacuated are placed when the respective proxy objects are in the train-managed generation being the trains to which the respective proxy objects belong or to which the respective proxy objects are currently destined for evacuation.

First Hit Fwd Refs

□ Generate Collection Print

L15: Entry 3 of 7 File: USPT Aug 13, 2002

DOCUMENT-IDENTIFIER: US 6434577 B1
** See image for Certificate of Correction **
TITLE: Scalable-remembered-set garbage collection

Brief Summary Text (4):
The present invention is directed to memory management. It particularly concerns what has come to be known as "garbage collection."

Brief Summary Text (16):
A way of reducing the likelihood of such leaks and related errors is to provide memory-space reclamation in a more-automatic manner. Techniques used by systems that reclaim memory space automatically are commonly referred to as "garbage collection." Garbage collectors operate by reclaiming space that they no longer consider "reachable." Statically allocated objects represented by a program's global variables are normally considered reachable throughout a program's life. Such objects are not ordinarily stored in the garbage collector's managed memory space, but they may contain references to dynamically allocated objects that are, and such objects are considered reachable. Clearly, an object referred to in the processor's call stack is reachable, as is an object referred to by register contents. And an object referred to by any reachable object is also reachable.

Brief Summary Text (18):
Garbage-collection mechanisms can be implemented by various parts and levels of a computing system. One approach is simply to provide them as part of a batch compiler's output. Consider FIG. 2's simple batch-compiler operation, for example. A computer system executes in accordance with compiler object code and therefore acts as a compiler 20. The compiler object code is typically stored on a medium such as FIG. 1's system disk 17 or some other machine-readable medium, and it is loaded into RAM 14 to configure the computer system to act as a compiler. In some cases, though, the compiler object code's persistent storage may instead be provided in a server system remote from the machine that performs the compiling. The electrical signals that carry the digital data by which the computer systems exchange that code are exemplary forms of carrier waves transporting the information.

Brief Summary Text (19):
The input to the compiler is the application source code, and the end product of the compiler process is application object code. This object code defines an application 21, which typically operates on input such as mouse clicks, etc., to generate a display is or some other type of output. This object code implements the relationship that the programmer intends to specify by his application source code. In one approach to garbage collection, the compiler 20, without the programmer s explicit direction, additionally generates code that automatically reclaims unreachable memory space.

Brief Summary Text (20):
Even in this simple case, though, there is a sense in which the application does not itself provide the entire garbage collector. Specifically, the application will typically call upon the underlying operating system's memory-allocation functions. And the operating system may in turn take advantage of various hardware that lends

itself particularly to use in garbage collection. So even a very simple system may disperse the garbage-collection mechanism over a number of computer-system layers.

Brief Summary Text (21):
To get some sense of the variety of system components that can be used to implement garbage collection, consider FIG. 3's example of a more complex way in which various levels of source code can result in the machine instructions that a processor executes. In the FIG. 3 arrangement, the human applications programmer produces source code 22 written in a high-level language. A compiler 23 typically converts that code into "class files." These files include routines written in instructions, called "byte codes" 24, for a "virtual machine" that various processors can be configured to emulate. This conversion into byte codes is almost always separated in time from those codes' execution, so FIG. 3 divides the sequence into a "compile-time environment" 25 separate from a "run-time environment" 26, in which execution occurs. One example of a high-level language for which compilers are available to produce such virtual-machine instructions is the Java.TM. programming language. (Java is a trademark or registered trademark of Sun Microsystems, Inc., in the Unites States and other countries.)

Brief Summary Text (23):
In most implementations, much of the virtual machine's action in executing these byte codes is most like what those skilled in the art refer to as "interpreting," and FIG. 3 shows that the virtual machine includes an "interpreter" 28 for that purpose. The resultant instructions typically invoke calls to a run-time system 29, which handles matters such as, loading new class files as they are needed and, of particular interest in the present connection, performing garbage collection.

Brief Summary Text (25):
The arrangement of FIG. 3 differs from FIG. 2 in that the compiler 23 for converting the human programmer's code does not contribute to providing the garbage-collection function; that results largely from the virtual machine 27's operation. Although the FIG. 3 arrangement is a popular one, it is by no means universal, and many further implementation types can be expected. Proposals have even been made to implement the virtual machine 27's behavior in a hardware processor, in which case the hardware itself would provide some or all of the garbage-collection function.

Brief Summary Text (26):
In short, garbage collectors can be implemented in a wide range of combinations of hardware and/or software. As is true of most of the garbage-collection techniques described in the literature, the invention to be described below is applicable to most such systems.

Brief Summary Text (27):
By implementing garbage collection, a computer system can greatly reduce the occurrence of memory leaks and other software deficiencies in which human programming frequently results. But it can also have significant adverse performance effects if it is not implemented carefully. To distinguish the part of the program that does "useful" work from that which does the garbage collection, the term mutator is sometimes used in discussions of these effects; from the collector's point of view, what the mutator does is mutate active data structures' connectivity.

Brief Summary Text (28):
Some garbage-collection approaches rely heavily on interleaving garbage-collection steps among mutator steps. In one type of garbage-collection approach, for instance, the mutator operation of writing a reference is followed immediately by garbage-collector steps used to maintain a reference count in that object's header, and code for subsequent new-object storage includes steps for finding space occupied by objects whose reference count has fallen to zero. Obviously, such an

approach can slow mutator operation significantly.

Brief Summary Text (29):
Other approaches therefore interleave very few garbage-collector-related instructions into the main mutator process but instead interrupt it from time to time to perform garbage-collection cycles, in which the garbage collector finds unreachable objects and reclaims their memory space for reuse. Such an approach will be assumed in discussing FIG. 4's depiction of a simple garbage-collection operation. Within the memory space allocated to a given application is a part 40 managed by automatic garbage collection. In the following discussion, this will be referred to as the "heap," although in other contexts that term refers to all dynamically allocated memory. During the course of the application's execution, space is allocated for various objects 42, 44, 46, 48, and 50. Typically, the mutator allocates space within the heap by invoking the garbage collector, which at some level manages access to the heap. Basically, the mutator asks the garbage collector for a pointer to a heap region where it can safely place the object's data. The garbage collector keeps track of the fact that the thus-allocated region is occupied. It will refrain from allocating that region in response to any other request until it determines that the mutator no longer needs the region allocated to that object.

Brief Summary Text (31):
A typical approach to garbage collection is therefore to identify all reachable objects and reclaim any previously allocated memory that the reachable objects do not occupy. A typical garbage collector may identify reachable objects by tracing references from the root set 52. For the sake of simplicity, FIG. 4 depicts only one reference from the root set 52 into the heap 40. (Those skilled in the art will recognize that there are many ways to identify references, or at least data contents that may be references.) The collector notes that the root set points to object 42, which is therefore reachable, and that reachable object 42 points to object 46, which therefore is also reachable. But those reachable objects point to no other objects, so objects 44, 48, and 50 are all unreachable, and their memory space may be reclaimed.

Brief Summary Text (32):
To avoid excessive heap fragmentation, some garbage collectors additionally relocate reachable objects. FIG. 5 shows a typical approach. The heap is partitioned into two halves, hereafter called "semi-spaces." For one garbage-collection cycle, all objects are allocated in one semi-space 54, leaving the other semi-space 56 free. When the garbage-collection cycle occurs, objects identified as reachable are "evacuated" to the other semi-space 56, so all of semi-space 54 is then considered free. Once the garbage-collection cycle has occurred, all new objects are allocated in the lower semi-space 56 until yet another garbage-collection cycle occurs, at which time the reachable objects are evacuated back to the upper semi-space 54.

Brief Summary Text (34):
In one sense, the approach of interrupting the mutator occasionally for garbage collection can increase an application's responsiveness, because the main mutator operation ordinarily proceeds relatively unburdened by garbage-collection overhead. In interactive systems, moreover, interruptions for garbage collection can sometimes be scheduled opportunistically so as to reduce the likelihood that they will result in much overall speed reduction. Garbage collection can be triggered when the system is waiting for user input, for instance.

Brief Summary Text (35):
So it may often be true that the garbage-collection operation's effect on performance can depend less on the total collection time than on when collections actually occur. But another factor that often is even more determinative is the duration of any single collection cycle, i.e., how long the mutator must remain

quiescent at any one time. In an interactive system, for instance, a user may never notice hundred-millisecond interruptions for garbage collection, whereas most users would find interruptions lasting for two seconds to be annoying. Many garbage collectors therefore operate incrementally. That is, they perform less than a complete collection in any single interruption of the main application.

Brief Summary Text (40):
Of course, the card-table approach is only one of many that can be employed to detect inter-generational pointers. Indeed, it is typical for an individual garbage collector to use more than one approach. Although there is no reason in principle to favor any particular number of generations, and although FIG. 6 shows three, most generational garbage collectors have only two generations, of which one is the young generation and the other is the mature generation. Moreover, although FIG. 6 shows the generations as being of the same size, a more-typical configuration is for the young generation to be considerably smaller. Finally, although we assumed for the sake of simplicity that collection during a given cycle was limited to only one generation, a more-typical approach is actually to collect the whole young generation at every cycle but to collect the mature one less frequently.

Brief Summary Text (41):
To collect the young generation, it is preferable to employ the card table to identify pointers into the young generation; laboriously scanning the entire mature generation would take too long. On the other hand, since the young generation is collected in every cycle and can therefore be collected before mature-generation processing, it takes little time to scan the few remaining, live objects in the young generation for pointers into the mature generation in order to process that generation. For this reason, the card table will typically be so maintained as only to identify the regions occupied by references into younger generations and not into older ones.

Brief Summary Text (42):
Now, although it typically takes very little time to collect the young generation, it may take more time than is acceptable within a single garbage-collection cycle to collect the entire mature generation. So some garbage collectors may collect the mature generation incrementally; that is, they may perform only a part of the mature generation's collection during any particular collection cycle. Incremental collection presents the problem that, since the generation's objects that are outside a collection cycle's collection set are not processed during that cycle, any such objects that are unreachable are not recognized as unreachable, so collection-set objects to which they refer tend not to be, either.

Brief Summary Text (73):
If the object remains popular, moreover, the number of entries that must thereafter be made to its car's remembered set during later collection cycles will also be high. This causes such remembered sets to become large and unwieldy. In the worst case, such an object can be referred to by almost every object in the generation, in which case the remembered set would be on the order of the entire generation's size. This not only causes a significant space problem but also makes maintaining and processing remembered sets costly. As a remembered set's size increases, the cost of adding new entries, eliminating duplicate entries for the same references, and scanning the references during the car's collection can become unacceptable. The garbage-collection overhead thereby imposed by popular-object-using applications may be so great as to make it impractical to provide garbage-collection intervals that are short enough to meet performance requirements.

Drawing Description Text (6):
FIG. 4 is a diagram that illustrates a basic garbage-collection mechanism;

Drawing Description Text (7):
FIG. 5 is a similar diagram illustrating that garbage-collection approach's

relocation operation;

Detailed Description Text (19):
After the card table is processed and the remembered sets updated, any younger generations are collected. For the sake of example, let us assume that the garbage-collected heap is organized into two generations, namely, a young generation, which is completely collected during every collection cycle, and a mature generation, which is collected incrementally in accordance with the train algorithm. The young generation's collection may involve promotion of some objects into the mature generation, and a new train will typically be allocated for this purpose, unless an empty train already exists. When the young generation's collection is completed, collection of the generation organized in accordance with the train algorithm will begin, and at this point a new train will typically be allocated, too, unless the newest existing train is already empty.

Current US Original Classification (1):
707/206

Other Reference Publication (1):
Moss et al. "A Complete and Coarse-Grained Incremental Garbage Collection for Persisten Object Stores" Proceedings 7th International Workshop on Persistent Object System, Cape May, NJ 1996, pp. 1-13.*

Other Reference Publication (4):
Liskov et al. "Partitioned Garbage Collection of a Large Stable Heap", Proceedings of IWOOOS 1996, pp. 117-121.*

Other Reference Publication (5):
Bill Venners, Garbage Collection, Inside the Java 2 Virtual Machine, Chapter 9, parts 1-18, www.artima.com., May 23, 2000.

Other Reference Publication (6):
Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", in Yves Bekkers and Jacques Cohen, editors, Proceedings of International Workshop on Memory Management, vol. 637 of Lecture Notes in Computer Science, 1992 Springer-Verlag.

Other Reference Publication (9):
Andrew W. Appel, "Simple Generational Garbage Collection And Fast Allocation", Software Practice and Experience, 19(2):171-183, 1989.

Other Reference Publication (10):
Richard Hudson and Amer Diwan, "Adaptive Garbage Collection For Modula-3 And Small Talk" in OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, Oct. 1990, edited by Eric Jul and Niels-Christian Juul.

Other Reference Publication (14):
Urs Holzle, "A Fast Write Barrier For Generational Garbage Collectors" in OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object Oriented Systems, Oct. 1993, edited by Moss, Wilson and Zorn.

Other Reference Publication (15):
Antony L. Hosking and Richard L. Hudson, "Remembered Sets Can Also Play Cards" in OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems, Oct. 1993, edited by Moss, Wilson and Zorn.

Other Reference Publication (16):
Jacob Seligmann and Steffen Grarup, "Incremental Mature Garbage Collection Using Train Algorithm", In The European Conference on Object-Oriented Programming 1995 Proceedings. Available at http://www.daimi.aau.dk/jacobse/Papers/.

Other Reference Publication (17):
Steffen Grarup and Jacob Seligmann, "Incremental Mature Garbage Collection" M.Sc.Thesis, avaliable at http://www.daimi.aau.dk/jacobse/Papers/. August 1993.

CLAIMS:

1. A method of garbage collection that employs remembered sets, associated with respective memory sections, to specify the locations of references to objects contained in the memory sections with which those remembered sets are respectively associated, and in which, for each of at least a given remembered set: A) the method includes generating the given remembered set by successively adding to the remembered set entries that specify respective memory regions that contain references to objects in the memory section with which the remembered set is associated; and B) the entries initially placed in the given remembered set specify the respective regions with an initial, relatively fine granularity, and the granularities of at least some entries placed therein after the previously added entries meet a set of at least one granularity-change criterion specify the reference-containing regions with a coarser granularity.

First Hit Fwd Refs

End of Result Set

□ Generate Collection Print

L21: Entry 1 of 1 File: USPT Mar 3, 1998

DOCUMENT-IDENTIFIER: US 5724581 A
**** See image for Certificate of Correction ****
TITLE: Data base management system for recovering from an abnormal condition

Detailed Description Text (78):
Type of operation: Various data base operations such as "update", "generation", "deletion", etc. performed on a logical page.

Detailed Description Text (144):
The operation of the above described recovery process performed to reconstruct a data base when a system failure arises is explained below by referring to the flowchart in FIG. 15.

Current US Original Classification (1):
707/202

First Hit Fwd Refs

□ Generate Collection Print

L29: Entry 10 of 11 File: USPT Jan 11, 1994

DOCUMENT-IDENTIFIER: US 5278982 A
**** See image for Certificate of Correction ****
TITLE: Log archive filtering method for transaction-consistent forward recovery from catastrophic media failures

Brief Summary Text (5):
Recovery from secondary stable storage media failures is an important problem to computer systems. In some cases, media recovery must be done entirely from offline storage media (such as a tape archive). This is particularly necessary in small systems with a single disk stable store but may also be required to recover from larger system disasters such as machine room fires or natural disasters. Transaction-based systems such as database management systems require recovery of a data resource stored on stable media to an atomic or transaction-consistent state. Conventional data resource recovery algorithms are intended for crash recovery and assume that the entire recovery log is available, including that portion of the log conventionally stored on-line in stable storage media. Loss of the on-line recovery log through stable media failure will prevent transaction-consistent crash recovery. Forward recovery using conventional recovery algorithms requires the recovery log archive tape to include all recovery log records. This requirement for storing all log records is troublesome because of the substantial storage volume occupied by such a recovery log archive.

Brief Summary Text (11):
The non-volatile or stable version of the recovery log is stored on stable storage such as rotating magnetic media ("disk"). Such stable storage can be improved by maintaining two identical copies of the recovery log on different disks. These on-line stable storage log records are then occasionally copied to a cheaper and slower archive medium such as tape. The recovery log archive records may be discarded once the appropriate image copy (archive dumps) of the database is produced, making the earlier recovery log archive records moot.

Brief Summary Text (13):
The UNDO records of a recovery log provide information on how to undo changes performed by the transaction. The REDO records of a recovery log provide information on how to redo changes performed by the transaction. In Write-Ahead Logging (WAL) based systems such as ARIES, an updated database is written back to the same stable storage location from where it was read. The WAL protocol asserts that the recovery log records representing changes to some data must already be in stable storage before the changed data are allowed to replace the previous version of those data in stable storage. That is, the system is not permitted to write an updated data page to the stable storage version of the database until at least the UNDO records of the recovery log describing the page update actions have been first written to stable storage.

Brief Summary Text (14):
Since a transaction includes execution of an application-specified sequence of operations, it is initiated with a special BEGIN transaction operation and terminates with either a COMMIT operation or an ABORT operation. The COMMIT and ABORT operations are the key to providing atomicity, as is known in the art. Transaction status is also stored in the recovery log and no transaction can be

considered complete until its COMMIT status and all of its recovery log records are safely recorded on stable storage by forcing to disk all recovery log records up through the LSN of the most recent transaction COMMIT record. This permits a restart recovery procedure to recover any transactions that completed successfully but whose updated pages were not physically written to stable storage before system failure. This means that a transaction is not permitted to complete its COMMIT processing until all REDO records for that transaction have been written to stable storage.

Detailed Description Text (9):
Referring to FIG. 3B, notice that the recovery log proceeds from P/C point to P/C point along solid line 50 until the last log archive dump (LAD) 52. Thereafter, the log continues to accumulate in stable storage (dotted line at 52) but is not present in the log archive. Pseudo-crashes exemplified by P/C point 44 occur at intervals along recovery log line 50. At each P/C, a series 58 of PSEUDO-RECOVERY type records are written followed by an END-PSEUDO-RECOVERY type record 60, shown as short line (P/C log) segments in FIG. 3B. The most recent of these segments is labeled LAD, which indicates that a Log Archive Descriptor representing the most recent P/C point 44 has been written to the log archive. An important feature to note in FIG. 3B is that recovery log creation proceeds simultaneously with the pseudo-crash recovery log generation. In practice, the two sets (50 and 58) of log records are merged together in a single recovery log. Record series 50 and 58 are shown on separate lines in FIG. 3B for illustrative purposes only.

Detailed Description Text (12):
Thus, in a single forward pass over the log archive from image dump point 54 to most recent END-PSEUDO-RECOVERY record 60, the data resource image dump archive has been recovered to a transaction-consistent state for all transactions preceding the most recent P/C point 44 stored in the Log Archive Descriptor. Note that in the example shown in FIG. 3B, all transactions subsequent to the LAD timestamp 44 up until the disk crash point 62 are lost. However, this transaction loss interval is no worse than the loss interval incurred with a forward recovery using an unfiltered recovery log archive made at the same time. In this example, the log filtered archive contains no UNDO type records and only the few additional pseudo-crash log records illustrated.

Detailed Description Text (20):
Thus, as before, the data resource image dump 68 is recovered to a transaction-consistent state for transactions through the most recent P/C point 66. All transactions subsequent to P/C point 66 through the disk crash point 78 are lost. By requiring the second short reverse pass 72, most PSEUDO-RECOVERY type records as well as all UNDO records can be filtered from the log archive.

Current US Original Classification (1):
707/202